DEEP LEARNING FOR ULTRASOUND DATA-RATE REDUCTION

Master thesis submitted to

Signal Processing Laboratory (LTS5)
Electrical and Electronic Section
École Polytechnique Fédérale de Lausanne

Author: Yeray Sainz Lorenzo
Supervisor: Prof. Jean-Philippe Thiran
Assistants: Dimitris Perdios
Adrien Besson

February 19, 2019

Contents

List of Figures

List of Tables

1 Introduction

This master thesis aims to assess the performance that can be achieved when ultrasound (US) data is compressed by using deep learning. Recently, deep neural networks (DNN) have been used for image and video compression providing successful results; in particular, models implementing convolutional neural networks (CNN).

As a starting point, different implementations for image and video compression using CNNs have been reviewed [1]–[5], [7]. Once reviewed, a first implementation for grayscale images based in [1] has been implemented and tested. Once the network implemented has reached acceptable results for images, the same network has been used for US data. In this second case, the network has been trained and tested for different configurations/modifications on it.

1.1 Motivation

US devices generate a set of signals that are carried from a transducer probe to a computer for further processing in order to obtain images. Those signals are transmitted between both ends through a set of cables, making up a high capacity data transmission channel.

In order to achieve a portable US device, in will be required to transfer the data through a much lower capacity channel e.g.USB cable or wireless, as seen in Figure 1.1.

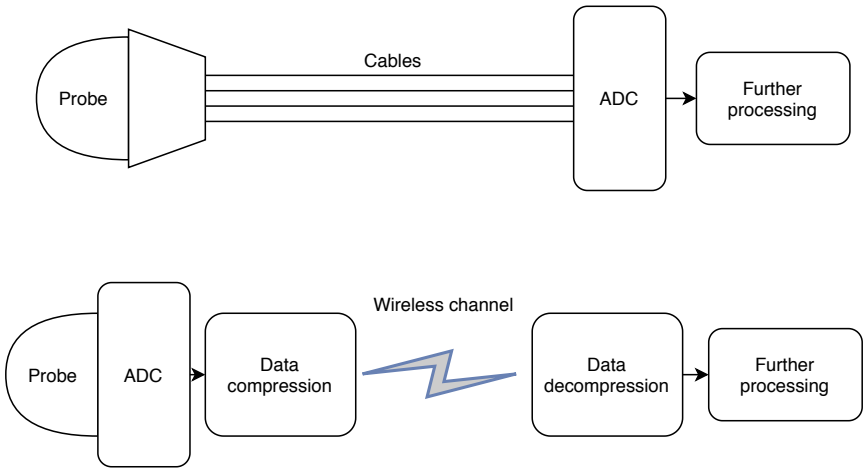

Figure 1.1 The usage of a low capacity channel to transmit the data will imply the usage of data compression

Therefore, a feasible way to transmit the data through a low capacity channel is through data compression. The models that have been implemented by different research groups for image and video compression purposes contain similar characteristics, which are presented within chapter 2.

1.2 Chapters summary

In chapter 2, they main ideas under data compression with autoencoder CNN architectures are explained. Followed by the objective function to optimize when training the neural networks, as well as the concept of rate-distortion trade-off that is set while training the model. The autoencoder CNN is introduced, as well as the common operations/layers within the encoding and decoding parts of the network. Furthermore, the key aspects to reach large compression ratios, which include quantization and entropy coding are also presented.

In chapter 3, an overview of the API used to develop the model is presented, as well as the main network architecture that has been used to obtain the results. It is also detailed how the rate and distortion terms explained in Section 2.2 are implemented in order to perform the trainings within Section 3.3. The description of other blocks of the model that pre-process and post-process the data, as well as the quantizer and entropy coder are also defined within Section 3.2.1.

In chapter 4, the training setup and metrics used to train the networks are defined. Followed by the results obtained from training and testing the network on images in Section 4.3. Tests performed on US data are shown in Section 4.4; in this case, the network has been modified in different ways in order to assess the compression performance when its complexity is reduced (mainly by reducing the number of layers and channels), in order to target real-time applications. The resulting quality in images obtained from the decompressed US data is shown and commented in Section 4.4.1. To end this chapter, in Section 4.4.2 some data rates are computed and compared with the capacity of USB and wireless channels.

At chapter 5, the conclusions obtained from the results as well as the future perspectives are commented.

2 Theoretical background

In this chapter the theoretical background lying under the compression with CNNs is explained, including the model overview, objective function to optimize, and CNNs. Other parts of the model that allow reaching larger compression ratios such as quantization and entropy coding in Section 2.4 are also introduced.

2.1 Model overview

From a high-level perspective, CNNs for image/data compression are based in an autoencoder structure plus a quantizer and lossless compression/decompression blocks as shown in Figure 2.1.

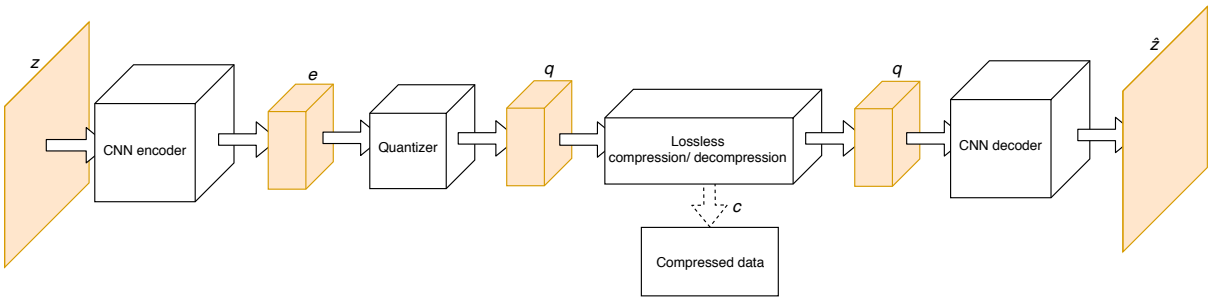

Figure 2.1 High level architecture of the data compression model, used for image and US data compression

The input data in Figure 2.1, named z, may be of dimensions $X \times Y \times Z$. X and Y may represent the pixels along the two dimensions of a grayscale image, or analogously, the number of probes X by sample values in time Y if it is US data. There will initially be a single channel (that is, Z will be 1 at both input z and output $\hat{z}$). Note that if instead of grayscale images, RGB images are used, Z will take value three to represent the three channels of a coloured image. During the work performed, only grayscale images have been used. The reason of not working with RGB images is that US data will consist, as well as grayscale images, in just one channel.

The input data z flows through the CNN encoder, where the relevant information of the data is extracted. In general, its spatial dimensions X and Y are reduced through the layers, while the number of channels Z increases. The resulting number of output data features at the CNN encoder output e, will be smaller that the number of input features.

The values at the output of the encoder e are then quantized to obtain q. The reason to perform this quantization step is to achieve larger compression rates in the following module of the model, the lossless compression step.

The lossless compression (and decompression) consists in an entropy coder. An entropy coder takes advantage of the quantized data q information entropy, obtained from Equation (2.4). The lower the entropy of q is, the larger the compression rate it will be achieved.

Eventually, the quantized values q that have been decompressed losslessly, flow through the CNN

decoder, as the number of channels Z is progressively reduced. Meanwhile, the spatial dimensions X and Y increase up to the original size of the input data. The CNN decoder output data $\hat{z}$ will be as close as possible to the original one, but including some distortions produced by the dimensionality reduction in the encoding process, as well as from the quantization step.

2.2 Objective function and rate-distortion trade-off

The model presented in Section 2.1 is end-to-end trained, as in [1]–[3]. Therefore, there is an objective function that includes the rate R and distortion D terms, plus a trade-off parameter β.

2.2.1 Objective function

From a high level point of view, the function to optimize for a given trade-off value β, is at Equation (2.1).

$$R + \beta D \tag{2.1}$$

The rate term R will be defined in terms of the quantized values' entropy (see Equation (2.4)), and the distortion term as $d(z, \hat{z})$. Resulting in Equation (2.2).

$$H[q] + \beta d(z, \hat{z}) \tag{2.2}$$

The trade-off parameter β does not have any intrinsic meaning; it is used to adjust the weight of the distortion loss, and converge into a given point, as explained in Section 2.2.2.

There are different approaches to model the entropy term while training the models; the model implemented (as in [1]) can be found in chapter 3.

2.2.2 Rate-distortion trade-off

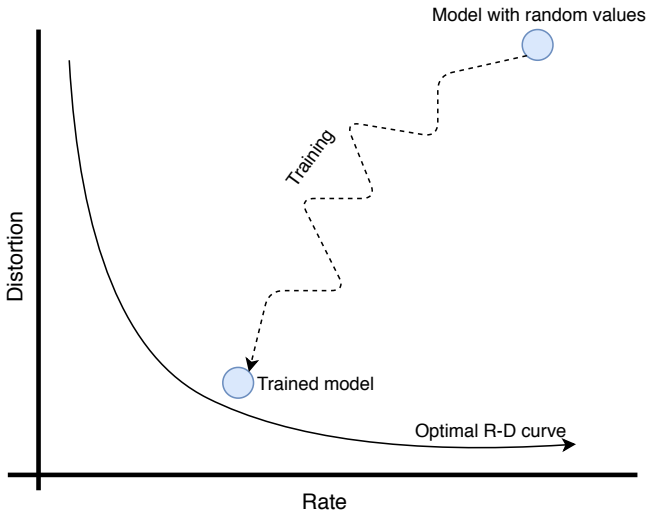

Figure 2.2 Optimization of the rate and distortion terms of the objective function, converging into a given point at the end of the training

The model is trained to optimize the objective function in Equation (2.1). By setting different values of β, different rate-distortion trade-offs will be achieved after each training. The purpose of the trade-off parameter β is mapping a point of the convex curve, as explained in [2]. As shown in Figure 2.2, the loss value will end up converging in a given point, which cannot be assured to be the optimal one. Using a

better optimizer or different hyperparameters might lead to a better final result.

2.3 CNN encoder/decoder

The autoencoder structure is composed by an encoder CNN and a decoder CNN. The encoder CNN extracts the relevant information from the input data into a smaller dimension, while the decoder CNN reconstructs the encoders' output data (which may have been previously quantized) into a result as close as possible as the original data, as shown in Figure 2.3.

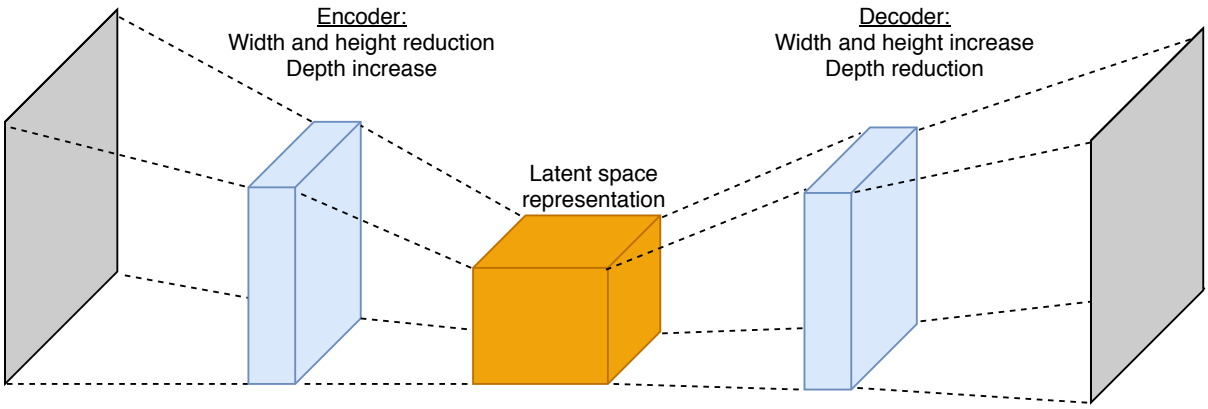

Figure 2.3 The encoder CNN reduces the width and height dimensions, while increasing the depth. The opposite behaviour for the decoder CNN

By tuning the strides, kernel sizes, filters, and padding, the desired output shapes through the encoder and decoder CNNs can be achieved, as explained with detail in [6].

The encoder CNN reduces the height and width and increases the depth of the input data. The common repeated structure within the encoder is a convolutional layer, followed by a downsampling layer to reduce the spatial dimensions (height and width), plus an activation function. The reduction of the spatial dimensions can be also achieved by setting the proper strides and padding (if any) values. This structure allows achieving a latent space representation of the input data.

In the decoder CNN, the encoder's output values have to be reconstructed into data as close as possible to the original one. Therefore, the opposite process has to be performed: increasing the height and width, and reducing the depth. Instead of convolutional and downsampling layers, transpose convolutions are used for this purpose, until reaching the dimensions of the original input data.

Convolutional layers

In convolutional layers, discrete convolutions are performed. In Figure 2.4, the following elements can be found; the blue grid is the input feature map, that is, the actual input data to be convolved by the convolutional layer. The green grid represents the result of the convolution. There is also a zero padding operation performed to this input feature map, increasing both width and height spatial dimensions by one, to obtain the desired output shape. The highlighted region in the padded input feature map represents the current position of the kernel during the convolution. The small-case values on it represent the kernel weights, which are the same all along the convolution. Those values are multiplied by the ones in the padded input feature map and summed, to obtain one of the values in the output feature map. In particular, the highlighted value in the convolution output (top left value), represents the result of such operation.

In this particular example, the stride of the convolution takes a value of two in both dimensions. That is, the kernel shifts two positions after each operation, as can be observed in Figure 2.5, Figure 2.6, Figure 2.7.

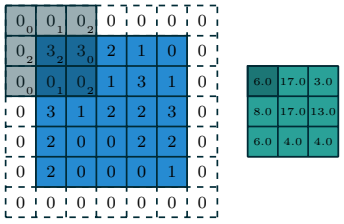

Figure 2.4 Convolution example with zero padding, kernel size of 3×3 and stride of 2×2. Image source [6]

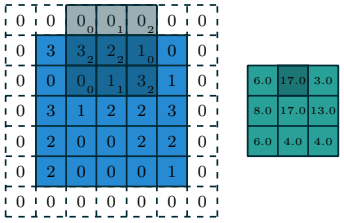

Figure 2.5 Convolution example with zero padding, kernel size of 3×3 and stride of 2×2. Image source [6]

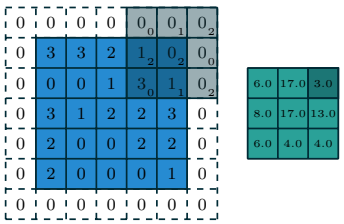

Figure 2.6 Convolution example with zero padding, kernel size of 3×3 and stride of 2×2. Image source [6]

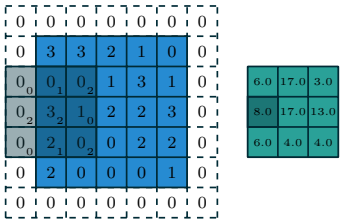

Figure 2.7 Convolution example with zero padding, kernel size of 3×3 and stride of 2×2. Image source [6]

Since in the example in Figure 2.4, Figure 2.5, Figure 2.6, Figure 2.7 just a single kernel is used, the output of the convolution consists in two dimensions. If a second kernel was used, to perform the same operation on the input feature map, the output would be three-dimensional. That is, instead of having a shape of 3×3, it would take a shape of $3 \times 3 \times 2$. In fact, using a second kernel in the example, would

imply reducing the spatial dimensions of width and height as well as increasing the depth, which is a typical convolution operation found in a encoder CNN.

Pooling operations

In is common to find a pooling operation performed on the convolutions' output values. The purpose of pooling operations is summarizing the content of a given region in a feature map, for instance, by taking the average value or the maximum value of such region. In Figure 2.8 and Figure 2.9, average polling and max pooling examples are shown. As it can be appreciated, the dimensions are reduced from 5×5 to 3×3.

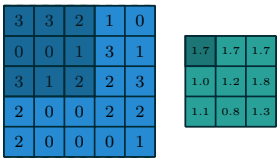

Figure 2.8 Average pooling example. Image source [6]

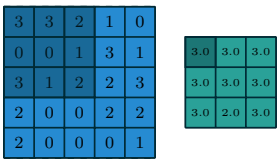

Figure 2.9 Max pooling example. Image source [6]

Transpose convolutions

In order to increase the width and height, transpose convolutions are used. These operations are used in the decoder CNN to sequentially increase the width and height, while reducing the depth of the features, to end up with an output of the same shape as the original input data. Note that if a convolution is performed, and then a transpose convolution operation is performed to the convolution's output data to obtain an output of the same dimensions as the original input data, the result will be of the same dimensions as the original input data, but it will not contain the same numerical values. Note that the operation to obtain exactly the same numerical values as the original ones, which is called deconvolution, is a different operation to a transpose convolution.

An example of a transposed convolution; consider that firstly a convolution operation has been performed, as the one in Figure 2.10, where from an input feature map of dimensions 4×4, an output of 2×2 is obtained. The transpose convolution that would revert the dimensions from 2×2 to 4×4 is the one in Figure 2.11. In this case, a zero padding of 2 (in general, of $kernel_size - 1$ in each of the kernel's dimensions) is performed in the 2×2 input feature map, to obtain the 4×4 output dimensions.

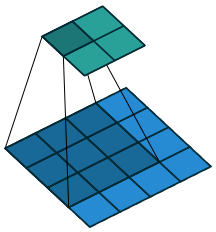

Figure 2.10 Convolution with no padding, unit strides, 3×3 kernel on a 4×4 input. Image source [6]

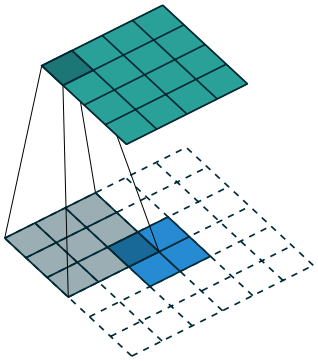

Figure 2.11 The transposed of convolving a 3×3 kernel on a 4×4 as in Figure 2.10. Image source [6]

In another convolution example where a stride of 2 is used, the padding of the transposed convolution should include zeros in between the values of the transposed convolution input. It is the case of the convolution in Figure 2.12 and transposed convolution in Figure 2.13. In this example, a convolution on a 5×5 input is performed with strides of value 2, resulting in an output of dimensions 2×2. Note that in the transpose convolution operation, which must be of dimensions 5×5, a stride of 1 is performed. However, this time the transposed convolution apart from being padded in the same way as in the previous example in Figure 2.11, zeros have been inserted in between the input values as well due to the stride value of 2 in the convolution operation.

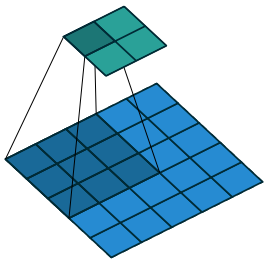

Figure 2.12 Convolution with no padding, strides of value 2, 3×3 kernel on a 5×5 input. Image source [6]

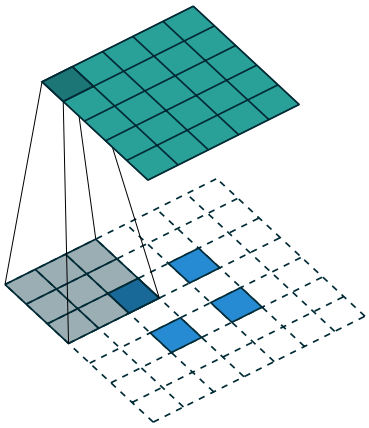

Figure 2.13 The transposed of convolving a 3×3 kernel on a 5×5 with strides of value 2, as in Figure 2.12. Image source [6]

Note that for other cases, such as padded convolutions, the way the padding is performed in the transposed convolution operations will be slightly different, but keeping many of the similarities of the two examples presented in Section 2.3.

It should be clarified that the examples presented in this Section 2.3, where several zero multiplications are performed due to the paddings, would be computationally inefficient. Nevertheless, the illustrations are used since are equivalent to applying the transposed convolution matrix $\mathbf{C}^\intercal$; considering that convolutions can be represented by a given convolution matrix $\mathbf{C}$. More detailed explanations and examples can be found in [6].

2.4 Quantization and entropy coding

As previously introduced in Section 2.1, the purpose of quantization is reducing the entropy of the encoder' s output values, see Equation (2.3). By performing the quantization step, the entropy of the encoder's output values will be reduced remarkably, as it can be observed in Figure 2.14.

$$q = Q(e) \tag{2.3}$$

The information entropy is obtained from Equation (2.4), where p_{x_i} represents the probability of a given value of q. By quantizing e, the total number of different values in q will be lower, as well as the entropy. This will imply a loss of information but a remarkable improvement in the final compression ratio, that will make up for that loss of information.

$$H = -\sum_i p_{x_i} \log_2 p_{x_i} \tag{2.4}$$

In Figure 2.14, it can be appreciated the effect of a rounding-to-the nearest-integer quantization in the entropy value. Note the difference in the ordinates axis; in the range $[-0.5, 0.5]$, the number of equal values before quantization did not overpass 50 in any case. After quantization, those values have been rounded to 0, contributing to reduce the entropy value. Values in other ranges also contribute to reduce the final entropy value, but the objective function's rate term of the implemented model to obtain the results in Figure 2.14, forces sparse coefficients, as in [1]. Therefore, the most common value after quantization will be zero.

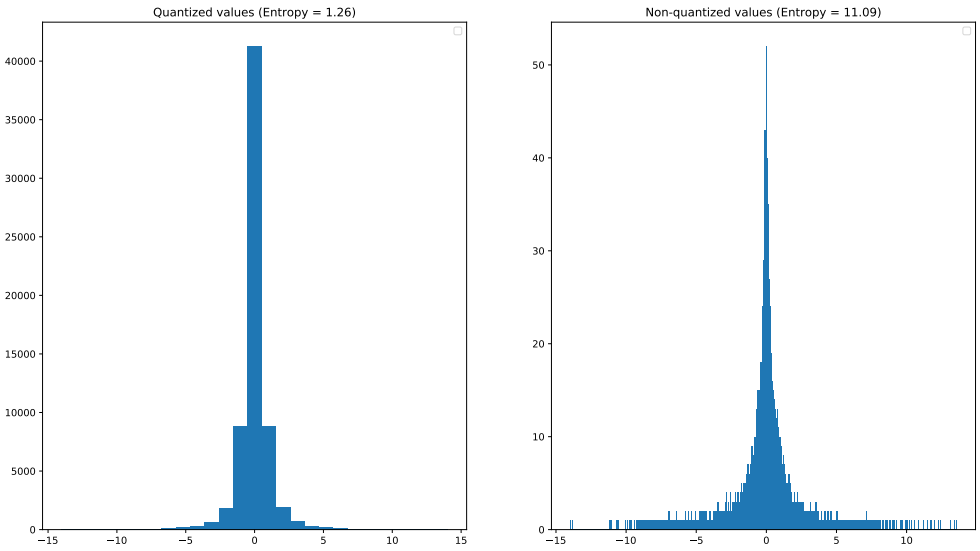

Figure 2.14 Histogram containing quantized and non-quantized values, plus the information entropy value corresponding to each case

2.4.1 Quantization step

Adding a quantizer during the training process after the encoder CNN, will make the gradients die at it during the back-propagation steps. The reason is that the derivative function of a quantization function will take value 0 or will not be defined, as can be observed in Figure 2.15. Therefore, it is necessary to find alternatives to overcome this issue.

In [7], quantization is replaced with additive uniform noise during training. This method outperforms the case where the effects of the quantization are ignored during training. See Equation (2.5), where u corresponds to uniform noise.

$$q \simeq e + u \tag{2.5}$$

Another alternative proposed by [1], [8], is quantizing during the forward pass in training, and passing the gradient values from the decoder CNN to the encoder CNN in the backward pass as if there was no quantization. This can be interpreted as considering in the backward pass a smooth approximation of the quantization function (e.g.rounding to the nearest integer), as it can be appreciated in Figure 2.15

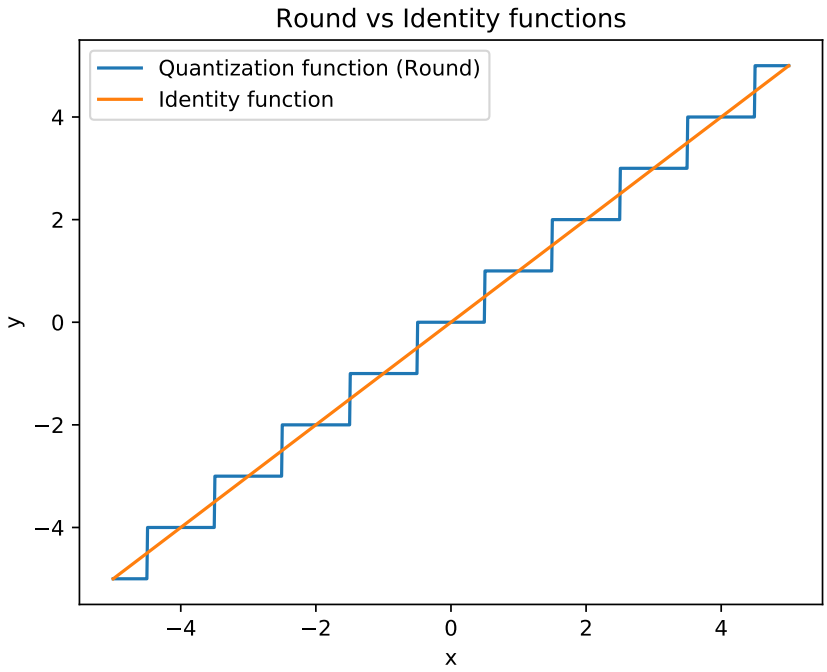

Figure 2.15 The identity function can be taken as a smooth and differentiable approximation of the rounding function in the backward pass

As in Figure 2.15, the identity function is considered in the backward pass (as if there was no quantization step). Therefore, the derivative of it will take value 1 as in Equation (2.6), and gradients will not be modified when passed from the decoder CNN to the encoder CNN.

$$\frac{d}{de}Q(e) \approx 1 \tag{2.6}$$

2.4.2 Entropy and range coding

The main idea under entropy coding is the following: Assigning short codes to the most frequent values, and long codes to the least frequent values. While the model is being trained, the rate term in the objective function is minimized. As a consequence, the entropy of the encoder's output values will decrease. As defined by Shannon's source coding theorem [9], the entropy of the quantized values will determine a lower bound for lossless compression methods.

As an example, consider the entropy values in the left graph of Figure 2.14, where the entropy value is 1.26 bits per symbol; In this case, the best possible lossless compressor will not be capable to compress all the values at a rate below 1.26 bits per symbol. If, for instance, there is a total of 10^6 values in the distribution of Figure 2.14 to be compressed, the lower bound for a lossless compressor will be 1.26 Mb.

Note that if quantization is not applied, which corresponds to the right graph in Figure 2.14, the entropy takes a value of 11.09. This means that the compression achieved would be about 10 times lower (worse) than in the case where the values are quantized.

The entropy coding method used in the final model implementation is range encoding, described in [10]. Range encoding can achieve compression rates which are very close to the information entropy. The way range encoding works is the following; the starting point before compression is a set of symbols (e.g.integer values), and their probabilities. From this point, a range of integer values is split into different sub-ranges, where each sub-range is proportional to the probability of each possible symbol, and the encoding algorithm is applied. The encoded/compressed data will be represented by a range of values

$[A, B)$. To decode/decompress the data, the decoder will also need to know the symbol probabilities to perform the decompression, by applying again the algorithm.

As an example; Consider that the set of symbols 'ABC' with probabilities shown in Table 2.1 wants to be compressed and an initial range of $[0, 10000)$ is taken, allowing for 10^4 possible symbol combinations. It is assumed that the decoder knows that 3 symbols will be encoded, as well as its probabilities. The set of subranges will be:

$$\begin{aligned} A &: [0, 8000) \\ B &: [8000, 9000) \\ C &: [9000, 10000) \end{aligned} \tag{2.7}$$

Once 'A' is encoded, the resulting sub-ranges are recomputed in the same way but this time within $[0, 8000)$, that is, A's sub-range. Therefore, the three sub-ranges computed will be the following ones:

$$\begin{aligned} A &: [0, 6400) \\ B &: [6400, 7200) \\ C &: [7200, 8000) \end{aligned} \tag{2.8}$$

The next symbol to be encoded is 'B'. The same procedure is performed, where this time the three sub-ranges will be contained in $[6400, 7200)$ resulting in:

$$\begin{aligned} A &: [6400, 7040) \\ B &: [7040, 7120) \\ C &: [7120, 7200) \end{aligned} \tag{2.9}$$

Eventually, 'C' is encoded; At this point, the resulting range of values that will uniquely identify the compressed sequence of symbols will be $[7120, 7200)$. Any of the 3-digit prefix values contained within the range $[712, 719]$ will uniquely identify the compressed sequence.

Table 2.1 Probability values for the different symbols in the range encoder example distribution.

Symbol	A	B	C
Probability	0.8	0.1	0.1

3 Implementation

In Section 3.1, a brief explanation of the API used to implement the model and how the code has been structured is given, followed by the implementation details of the different parts of the model. The network architecture, which has been based in [1], is also presented.

3.1 Software

The implementation has been developed in Python and TensorFlow; in particular, using the high-level API called Estimators shown in Figure 3.1. Mid-level APIs such as layers and metrics have also been used to develop the network architecture and to monitor the loss on a validation set during trainings respectively. To monitor the trainings, the TensorBoard visualization tool has been used.

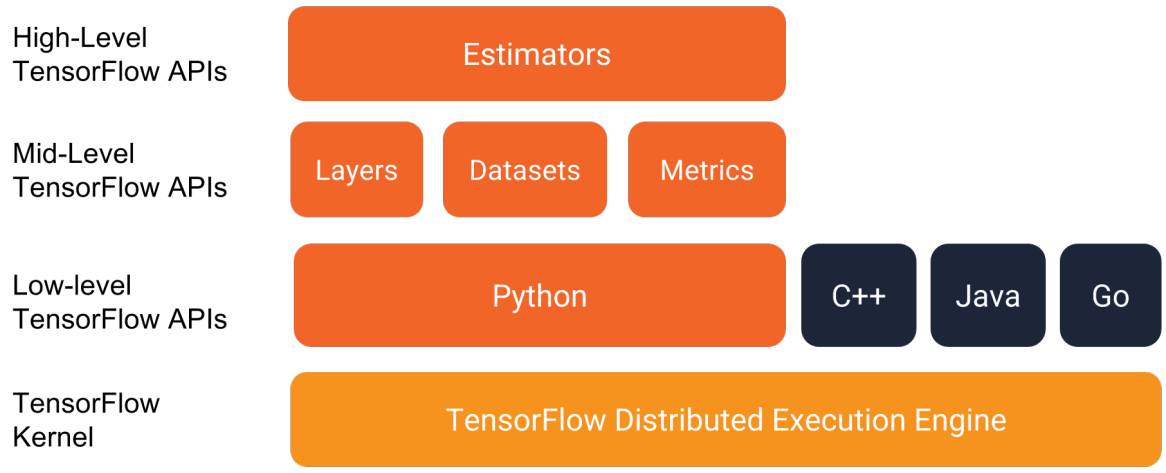

Figure 3.1 Programming stack showing the APIs that build the TensorFlow Estimators API. Source [16]

TensorFlow Estimators are composed of three methods: train, predict and evaluate. Firstly, an Estimator object is instantiated (Figure 3.2), passing as arguments a model function, a directory path where to save/load checkpoints from the trainings, and a list of parameters which include a list of rate-distortion trade-offs, a learning rate, an action to perform (encode, decode, train), and the number of steps (to train).

```
autoencoder_estimator = tf.estimator.Estimator(
    model_fn=model.autoencoder_model,
    model_dir=directory,
    params=parameters)
```

Figure 3.2 Estimator object, having as arguments a model function, a directory path, and a list of parameters

An input function is also defined (Figure 3.3), passing as arguments the input and the desired output values (which in this particular problem will be the same set of values), the batch size, and others.

```
train_fn = tf.estimator.inputs.numpy_input_fn(
                x={"x": train_dataset},
                y=train_dataset,
                batch_size=batch_size,
                num_epochs=None,
                shuffle=True
            )
```

Figure 3.3 Input function definition to perform a training with Estimators

Following this, one of the three methods from Estimators can be called and, among other arguments, the input function is passed. In Figure 3.4 the train method from Estimators is called.

```
autoencoder_estimator.train(input_fn=train_fn, steps=steps)
```

Figure 3.4 The train method is called to start a training of a given number of steps

Within the model function, which in this case has been custom built, three different cases are taken into account depending on which of the three methods from Estimators is called (train, predict or evaluate). If the predict mode is the method called, two sub-actions are defined: encode and decode. Which of these two functions is called can be set by the input parameters shown in Figure 3.2. The structure of the model function can be found in Figure 3.5.

```
def autoencoder_model(features, mode, params):
    if mode == tf.estimator.ModeKeys.TRAIN:
        return act.train(...)
    if mode == tf.estimator.ModeKeys.PREDICT:
        if params['action'] == 'encode':
            return act.encode(...)
        else:
            return act.decode(...)
    if mode == tf.estimator.ModeKeys.EVAL:
        return act.evaluate(...)
```

Figure 3.5 Pseudo-code of the custom model function defined

The methods act.train(), act.encode(), act.decode() and act.evaluate() are custom built to perform the specific instructions. Within those defined methods, another set of functions which include encoder(), decoder() and quantize() are called depending on the task to perform.

3.2 Network architecture

The network architecture can be found in Figure 3.6.

The network is composed from an encoder CNN (upper branch) and a decoder CNN (lower branch), plus some other blocks Figure 3.6. The main difference is that on the encoding branch, mirror padding and valid convolutions are performed. Both encoder and decoder CNNs include three residual blocks, composed by two convolutions of $f_2 \times 3 \times 3$ followed by a sum of the residual block's input plus the two convolutions' outputs, which will be of the same dimension. The architecture allows replacing the valid padding (no-padding) convolutions in the encoder CNN by same (zero-padded) convolutions; in this case, the mirror padding block will not perform any padding on the data.

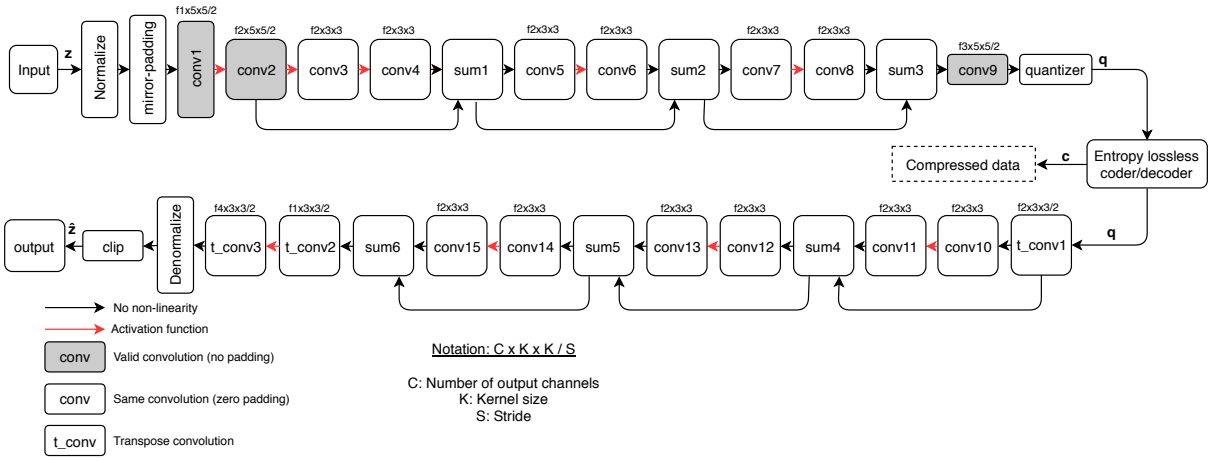

Figure 3.6 High level architecture of the data compression model, used for image and US data compression

3.2.1 Additional blocks

The additional blocks that process the data in different ways are the following:

Normalization and denormalization

The first pre-processing step performed on the data is normalization, by using the arithmetic mean and standard deviation values of the training set, as shown in Equation (3.1). Where μ and $sigma$ are the mean and standard deviation of the samples in the training dataset respectively, and x_i is each of the samples in the input feature map.

$$\frac{x_i - \mu}{\sigma} \tag{3.1}$$

By performing normalization (in particular, standardization), the input values will approximately have a mean of zero, and a standard deviation of 1. This normalization helps to improve the performance of the network, increasing the convergence speed during the trainings and reducing the output distortion/error.

Mirror padding

As in [1] network architecture, there is a block performing mirror padding to the normalized input features. The encoder CNN includes valid padding convolutions (that is, no padding). The width and height of the input features is reduced progressively through the layers, as the depth of the features increases. Since at the output of the encoder CNN it is desired to have a width and a height 8 times lower than the input one, the mirror padding operation is performed to obtain the desired output shape when applying valid convolutions at the encoder CNN. The concept of mirror padding is shown in Figure 3.7. In particular, a mirror padding of 11 values is performed in both width and height dimensions to obtain the desired output.

Quantizer

The quantization step is performed as a rounding to the nearest integer function, as in [1].

$$q = round(e) \tag{3.2}$$

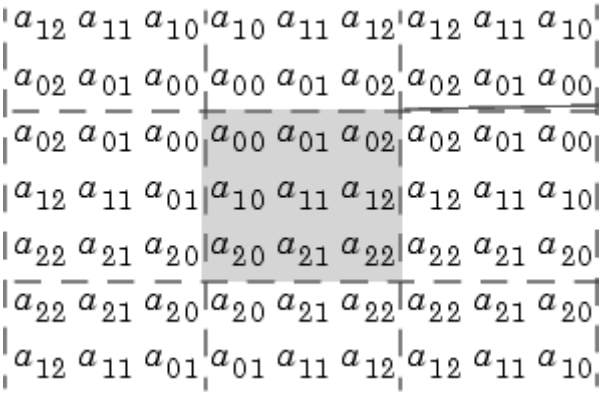

Figure 3.7 Mirror padding example. Source [12]

Clip function

The clip function ensures that the output data is within the desired range, by setting the values larger than the maximum expected value to the maximum expected value, and the values smaller than the minimum expected value to the minimum expected value.

Entropy lossless coder/decoder

The range coder used to losslessly compress and decompress the quantized values has been developed in `C++` by [13], and is used through a `Python` interface developed by [14].

The range coder computes the probabilities of each set of quantized values (being the value 0 always the most common). As it is shown in chapter 4, there are two possible alternatives to obtain the probability values. On one hand, the exact probability distribution of each test data batch to be compressed can be computed separately, to obtain the largest possible compression ratio for the implementation used; however, computing the probabilities for each batch of test data may take a relevant amount of computations. On the other hand, a probability distribution can be pre-defined, as it is done in [1]. This probability distribution can be obtained from Laplace-smoothing the histogram of the training set. The reason to apply smoothing is to remove the values with zero probability in the quantized values' cumulative density function used to perform the compression. That is, there will be some integer values in the probability distribution of the training set's quantized values that have zero probability, but which may appear on the test set. Even though it cannot be appreciated, on Figure 2.14 there are several values on the Gaussian-shaped histograms (particularly on the queues) that take value 0. Therefore, those values have to be set to a probability different from zero so that can be compressed properly by the range coder in case those values appear when using test data on the model.

Considering that the probability of each value x_i is defined as Equation (3.3), where p_{x_i} is the probability of the value x_i, and N is the total number of values.

$$p_{x_i} = \frac{x_i}{N} \tag{3.3}$$

The Laplace-smoothed probabilities are defined as Equation (3.4); where α is the pseudo-count, and takes value 1, and d is the number of different integer values in the range of the training set's quantized values.

$$p_{x_i\,smooth} = \frac{x_i + \alpha}{N + \alpha d} \tag{3.4}$$

3.3 Objective function implementation

The objective function introduced in Section 2.2.1 is composed of two terms, the rate R and the distortion D. In the implementation, both terms have been modelled as the mean squared error (MSE) for the distortion term, and Gaussian Scale Mixtures (GSM) for the rate term (as in [1]).

3.3.1 Distortion modelling

The distortion term is modelled as the MSE between the input and output data as in Equation (3.5). Where z_i is a given input feature, $\hat{z}_i$ the same feature at the output, and N the total number of input/output features.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(z_i - \hat{z}_i)^2 \tag{3.5}$$

3.3.2 Rate modelling

To model the distributions of coefficients, a mixture distribution is used as in [1]. In particular a mixture of Gaussian distributions. A mixture distribution $f(x)$ is the sum of weighted probability distributions (Equation (3.6)). There are a total of N distributions in the mixture , and each probability distribution $p_i(x)$ is weighted by a given weight π_i. The sum of the weights is equal to 1 (Equation (3.7)).

$$f(x) = \sum_{i=1}^{N}\pi_i p_i(x) \tag{3.6}$$

$$\sum_{i=1}^{N} w_i = 1 \tag{3.7}$$

The Gaussian distributions that compound the mixture (GSM) have zero mean, and the weights and variances are learned. For each set of values in each channel (depth dimension), six scales are used. That is, there are as many mixtures as channels, and each of the mixtures is constituted of six weighted Gaussian distributions.

The loss in the objective function resulting from the rate term is obtained from Equation (3.8).

$$-\sum_{i,j,k}\log_2\sum_{s}\hat{\pi_{ks}}\mathcal{N}(q_{ij}+u_{ij};0,\hat{\sigma_{ks}^2}) \tag{3.8}$$

For each channel k, the sum of the base 2 logarithms of each quantized value q_{ij} plus uniform noise u_{ij}, assessed in the mixture of the channel k where q_{ij} belongs to is computed. The minus sign is applied to eventually obtain a positive loss value, since the Gaussian mixtures will always take a value smaller than 1. The term s represents each of the Gaussian distributions per mixture. Note that all the Gaussian distributions of all the mixtures are centered; that is, have zero mean ($\mu = 0$).

The parametrization of a Gaussian mixture distribution, $\mathcal{N}(x;\mu,\sigma^2)$ is commonly composed from a set of mean values μ, a set of variances σ^2, and a set of weights π. During the first steps in the trainings, the coefficients' distribution will not be good to obtain a low entropy or a high compression (that is, the quantized coefficients' distribution will be quite uniform). As the training evolves, the encoder CNN will force a sparse distribution of the coefficients, in order to minimize the rate term loss contribution and therefore the entropy of the distribution (Same concept as the one in Section 2.4). Meanwhile, the Gaussian mixtures of each of the channels will also be adjusted to the varying quantized coefficients' distribution (until a given rate-distortion trade-off is reached).

To modify the Gaussian distributions during the training, the weights π_{ks} and variances σ_{ks}^2 are trained using gradient-based methods. Therefore, it is necessary to use another parametrization of log-weights $\hat{\pi_{ks}}$ and log-variances $\hat{\sigma_{ks}^2}$. These parametrizations are shown in Equation (3.9) and Equation (3.11).

$$\hat{\pi_s} = \sigma(\pi_s) \tag{3.9}$$

Where $\sigma()$ is the softmax function, defined in Equation (3.10). Therefore, the sum of the S weights π_s, will be one, as required by the mixture distributions constraint defined in Equation (3.7).

$$\sigma(\pi_s) = \frac{e^{\pi_s}}{\sum_k^S e^{\pi_k}} \tag{3.10}$$

The log-variances $\hat{\sigma_s^2}$ are defined in Equation (3.11), by computing the exponential of the variances σ_s^2.

$$\hat{\sigma_s^2} = e^{\sigma_s^2} \tag{3.11}$$

The parametrizations $\hat{\sigma_s^2}$ and $\hat{\pi_s}$ are mentioned in [1], but the specific formulation of them has been clarified personally by the author.

4 Tests and results

In this section the tests performed and results obtained both for grayscale images as well as for US data are shown. In addition, the qualities of the images obtained from the decompressed US data for different rate-distortion trade-offs is also analyzed. At the end, some expected data-rates are computed in order to assess if could be transmitted through USB and wireless channels with the current data transmission protocols.

4.1 Training setup

The optimizer used to perform the trainings is the TensorFlow implementation of Adam optimizer. As shown in [15], this optimizer has a better performance than other alternative gradient-based optimizers. Furthermore, this optimizer is the one used in all the reference papers dealing with image compression using neural networks [1]–[5].

Each training has been performed for a total of 750000 steps, even though the rate and distortion losses converged earlier (After approximately 500000 steps). Each step consists in a forward and a backward pass of a 32-sample (images or US data) batch of dimensions 128×128. The exponential rate decay is initially set to 10^{-5}, with decays every 50000 steps and a decay rate of 0.5, as shown in Equation (4.1).

$$LR = 10^{-5} 0.5^{\lfloor step/50000 \rfloor} \tag{4.1}$$

The number of filters (output channels) are in Table 4.1. Note that the S2 setup has a half of the filters in S1, rounded to the lower integer, and there is the same relation between S2 and S3.

Table 4.1 Different filters setups for the network in Figure 3.6

Setup	f1	f2	f3	f4
S1	21	42	32	1
S2	10	21	16	1
S3	5	10	8	1

4.2 Metrics

The metrics used to assess the results have been the Structural Similarity index (SSIM) and the Peak Signal to Noise Ratio (PSNR).

The SSIM, defined in [18], is an index used to measure the perceptual quality of images and videos. This index considers the degradation in images as the perceived changes in structural information. The

structural information can be understood as the strong dependencies between pixels of an image in a given region of an image. On the contrary, the PSNR takes into account the absolute differences between original and decompressed values. PSNR is defined as the ratio between the maximum power of a signal/image and the MSE between the original and decompressed values of that signal/image, usually presented in the logarithmic scale as in Equation (4.2).

$$PSNR = 10log_{10}(\frac{Pmax^2}{MSE}) \tag{4.2}$$

4.3 Tests on images

Firstly, the network implemented has been tried on grayscale images. The reason to use grayscale images instead of RGB images is that US data is composed of a single channel, as grayscale images. The training datasets used to train the network are composed from images of Flickr and from the ImageNet dataset. The images obtained have been cropped to dimensions of 128×128. To obtain the results/metrics, images from the Kodak dataset has been used [17]. The training and validation datasets from ImageNet are composed from 48000 and 3000 crops obtained from images respectively. Using a larger training dataset has not lead to better results. The trainings performed on crops from Flickr high resolution images (as in [1]), have not lead to better results.

The differences between the network implemented and the reference network are the usage of transposed convolutions instead of sub-pixel convolutions in the decoder CNN, and the reduction of the number of channels in all the layers by a factor of three, and rounded to the lower integer. Tests performed with the same number of channels as in [1] have not lead to better results. The network architecture is the same as the one in Figure 3.6.

The training performed has not lead to better results than the ones obtained by using JPEG 2000, but are not far from the ones obtained with JPEG 2000 in terms of the SSIM metric. The results obtained in [1] showed a better performance than in JPEG 2000; the worse performance obtained may be due to the fact that sub-pixel convolutions are not used in the implemented network and/or to the different way the network is trained (incremental training in [1]). The results obtained are in Figure 4.1.

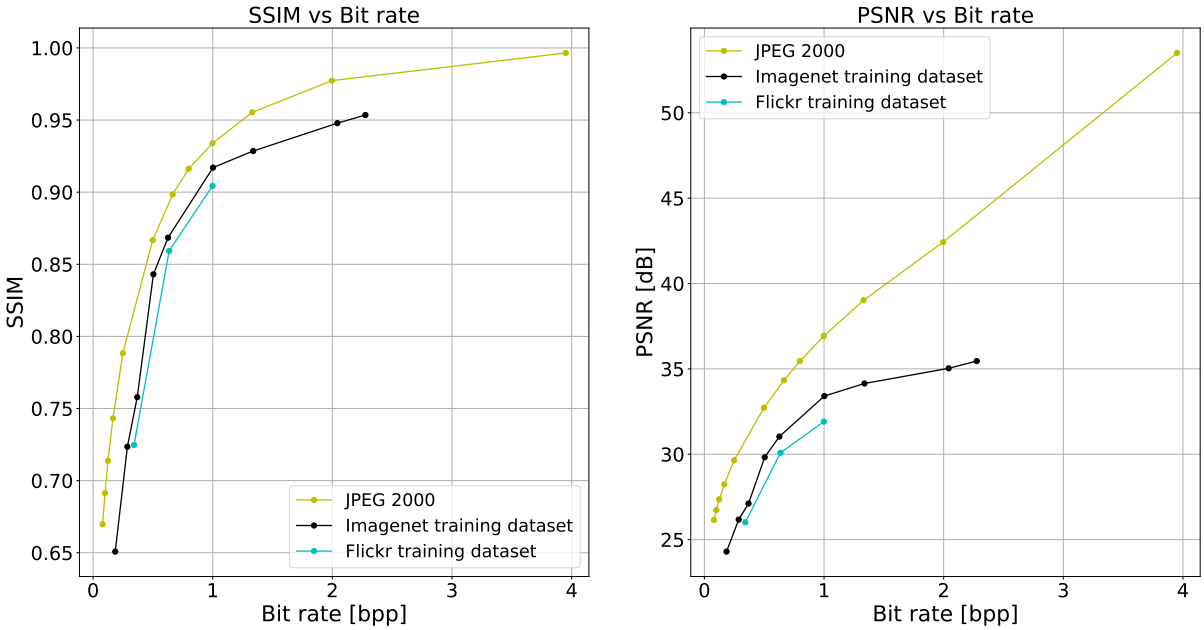

Figure 4.1 Metrics obtained from the ImageNet dataset, as well as from Flickr images

Even though the image obtained from Flickr were of high resolution (and compression artifacts were far less present than in the ImageNet dataset images), the results obtained from training the network with the Flickr training dataset have been slightly worse than the ones obtained with the ImageNet training dataset.

In Figure 4.2 and Figure 4.3 there is a comparison between compressed images from the network implemented and JPEG 2000. As shown, the results from using both methods are similar, for values over 0.5 bpp. Below of 0.5 bpp, the differences become far more relevant, as it can be observed from the case of 0.25 bpp, or a compression ratio of 32. The original images can be found in [17].

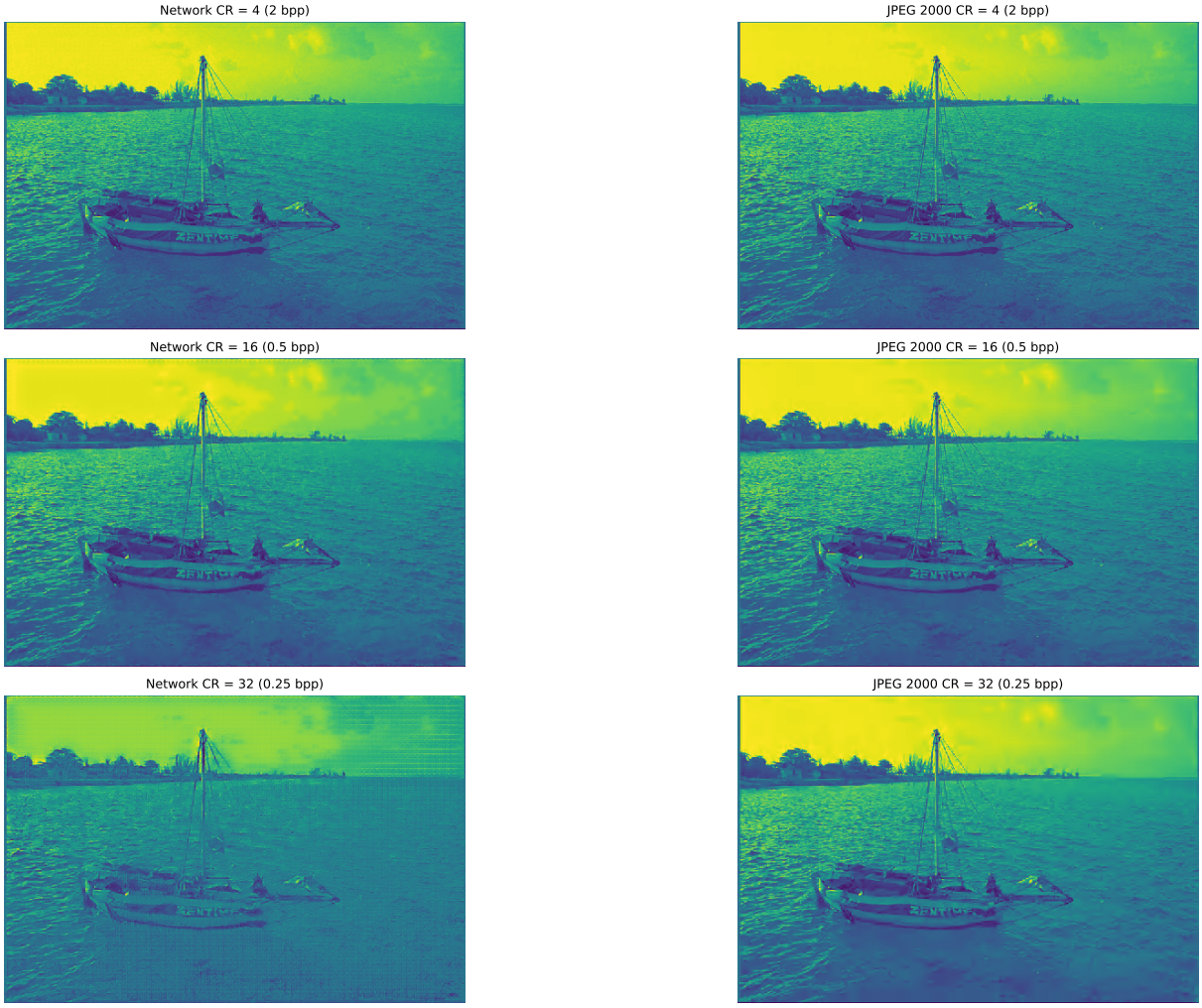

Figure 4.2 Image from the Kodak image dataset, compressed for different compression ratios

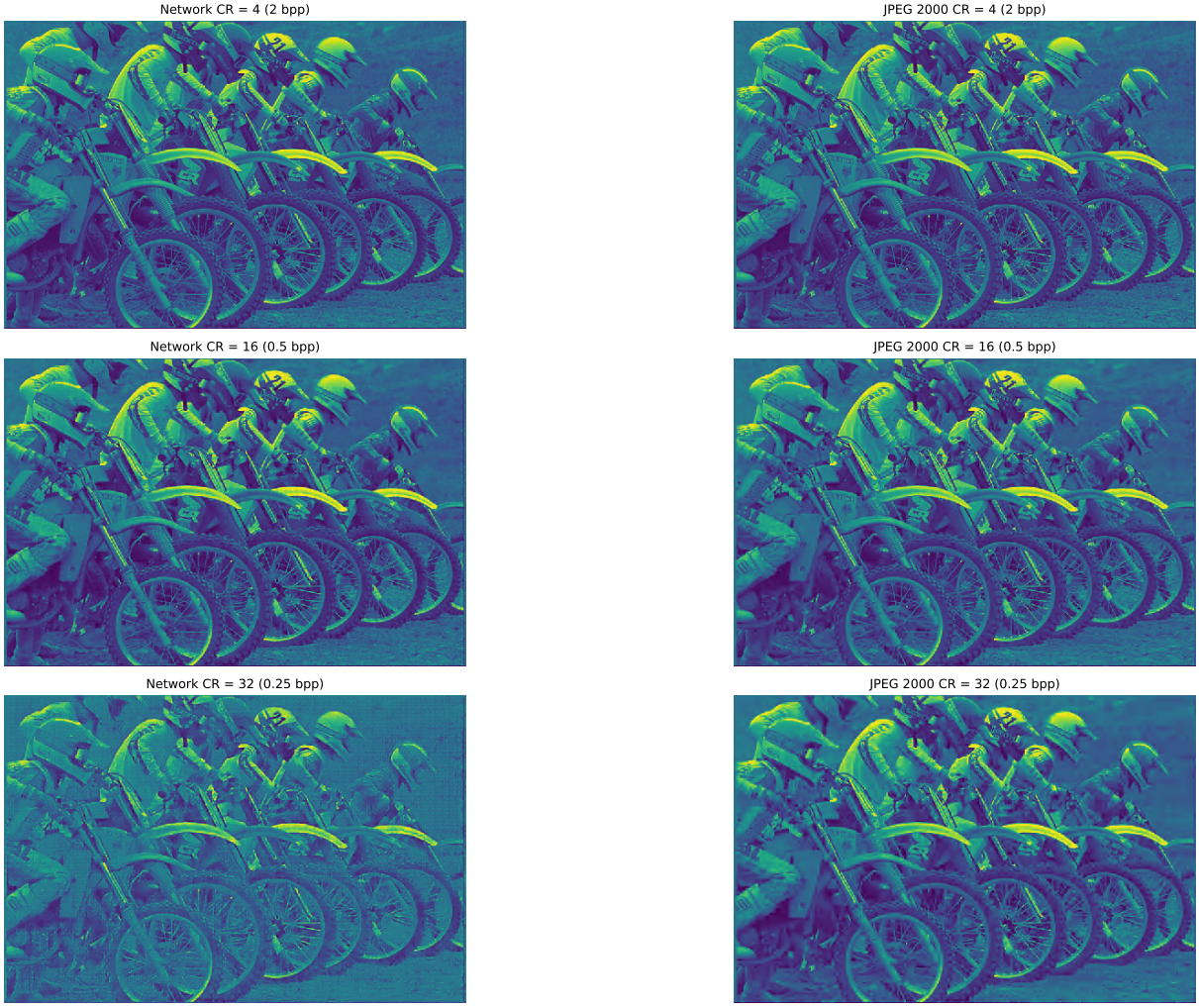

Figure 4.3 Image from the Kodak image dataset, compressed for different compression ratios

4.4 Tests on ultrasound data

To perform the trainings for US signals, the training and validation datasets used provide from simulated US signals, while the test datasets used provide from a high-dynamic-range simulated US signal, real US data from the carotid, and real *in vitro* US data. To obtain the training dataset, the data composed of 128×1024 values obtained from the PICMUS [19] dataset is cropped to obtain 200000 patches of 128×128. The results in this chapter are from the test dataset providing the worse results, which is the dataset with a high-dynamic-range. In figure Figure 4.9, all the test datasets are compared between each other. A difference with images is that each of the US values is composed of 2 bytes, instead of 1 byte. Furthermore, modifications to the network in Figure 3.6 have been performed to experiment with different configurations.

Activation functions: ReLu vs Leaky ReLu

Figure 4.4 shows the difference of using the ReLu function vs the leaky ReLu function, which is the one implemented in the original architecture in [1]. The difference between both functions is that in the negative domain, the leaky ReLu function is different from zero. In particular, it takes the following definition: $f(x) = 0.01x$ if $x < 0$. As it can be observed, the differences between the usage of ReLu and leaky ReLu are irrelevant for high SSIM values (over 0.9) and become more relevant for values of

SSIM below 0.8. The results from using JPEG 2000 are also shown, even though the model implemented outperforms it by obtaining from 2 to 3 times larger compression ratios for SSIM values larger than 0.9, and even larger larger compression ratios for SSIM values below 0.9.

Comparison between the number of channels

Figure 4.5 shows the comparison between different filters setups from Table 4.1. If the number of channels is reduced the maximum SSIM the network can provide on the US data for a given filters' setup is also reduced. For the common range of compression where the three setups have been assessed, the performance is similar in terms of SSIM. Values of SSIM below 0.9 will show remarkable compression artifacts after processing the US data and obtaining the final US images, therefore, the setup S3 may not be useful, since it is only capable to provide maximum SSIM values of approximately 0.83. Besides, the S2 setup contains the half of channels than S1, and shows very similar performance to S1 for SSIM and PSNR values over 0.9. For this reason, S2 may be more suitable since the inference time will be lower. If SSIM qualities of over 0.96 are the target, then a setup with more channels than S2 (such as S1) is needed.

Padding comparison

In the encoder CNN is it possible to perform mirror padding at the input feature map followed by non-padded (valid) convolutions, or not performing any mirror padding at the input feature map and then zero-pad before each convolution. From the results obtained in figure Figure 4.6, it can be seen that the differences between both possibilities are irrelevant, both in SSIM and PSNR terms.

Compression loss due to the cumulative density function (CDF) smoothing

As introduced in Section 3.2.1, the CDF used by the range encoder to losslessly compress the quantized values can be obtained from the training dataset, and then smoothed to be used on the test data. By using a smoothed CDF, a less efficient final compression is achieved than if a new CDF is computed for each set of test values. The advantage of using always the same CDF is that it will not be necessary to recompute the CDF of each set of values, reducing the total inference time. In Figure 4.7 there is a comparison between using the exact CDF and the Laplace-smoothed CDF from the training set. The effect of performing the Laplace-smoothing on the CDF is more remarkable for large compression ratios (e.g. when the quantized coefficients' distribution tends to be more sparse). The values obtained are from 500 US samples of 128×1024, generated in the same way as the training dataset. Note that the larger the amount of data to be jointly compressed is, the more similar its CDF will be to the Laplaced-smoothed one; therefore, better compression ratios may be obtained. Nevertheless, note that for low distortion (or equivalently, for a low compression ratio) there is almost no difference between the Laplace-smoothed and the exact CDF cases.

Residual blocks removal

The effect of removing the residual blocks on the network has been tested. As shown in Figure 4.8, the effect of removing the residual blocks on the test set has almost no effect in the final rate-distortion trade-off values.

Test datasets comparison

There is a relevant difference between using different datasets as shown in Figure 4.9. Note that in all the other figures, the high-dynamic-range is the one that has been used to obtain the metrics. Therefore, it is

expected a better performance of the network in the other test datasets for all the different configurations tested in the high-dynamic-range test dataset.

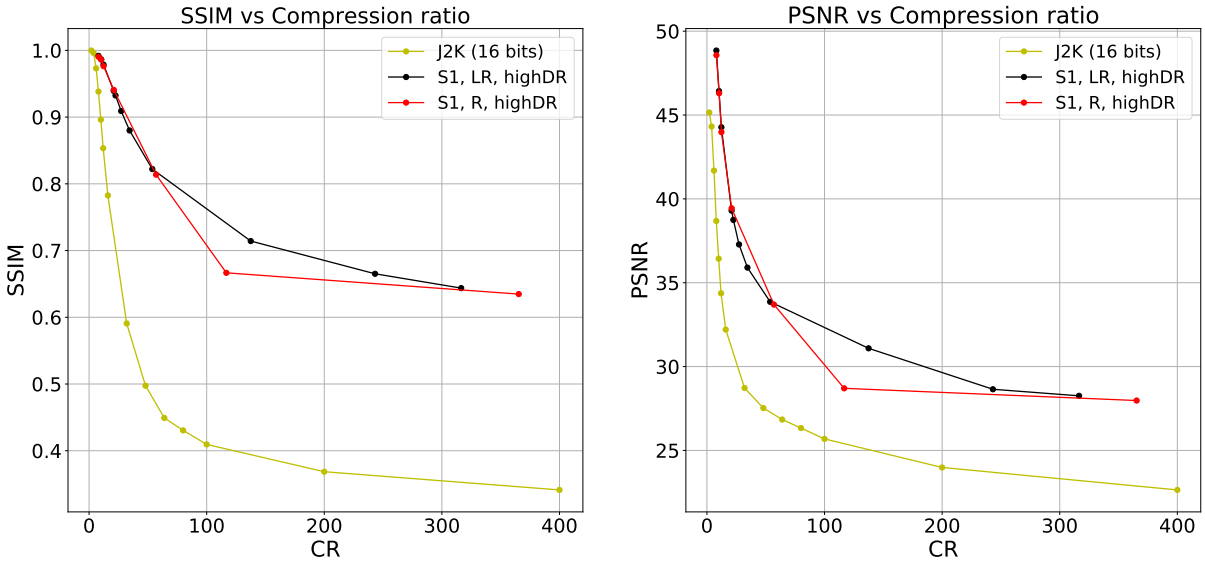

Figure 4.4 Comparative of the usage of ReLu and Leaky ReLu activation functions and S1 filters setup in the network and JPEG 2000

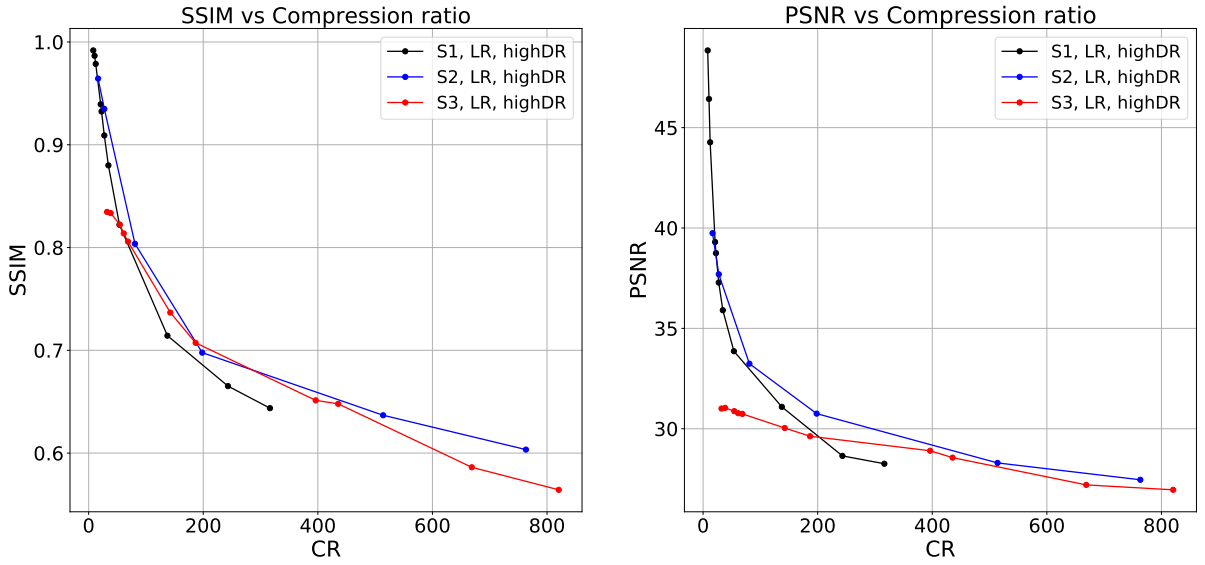

Figure 4.5 Comparative of the SSIM and PSNR metrics for different filters' setups on the network

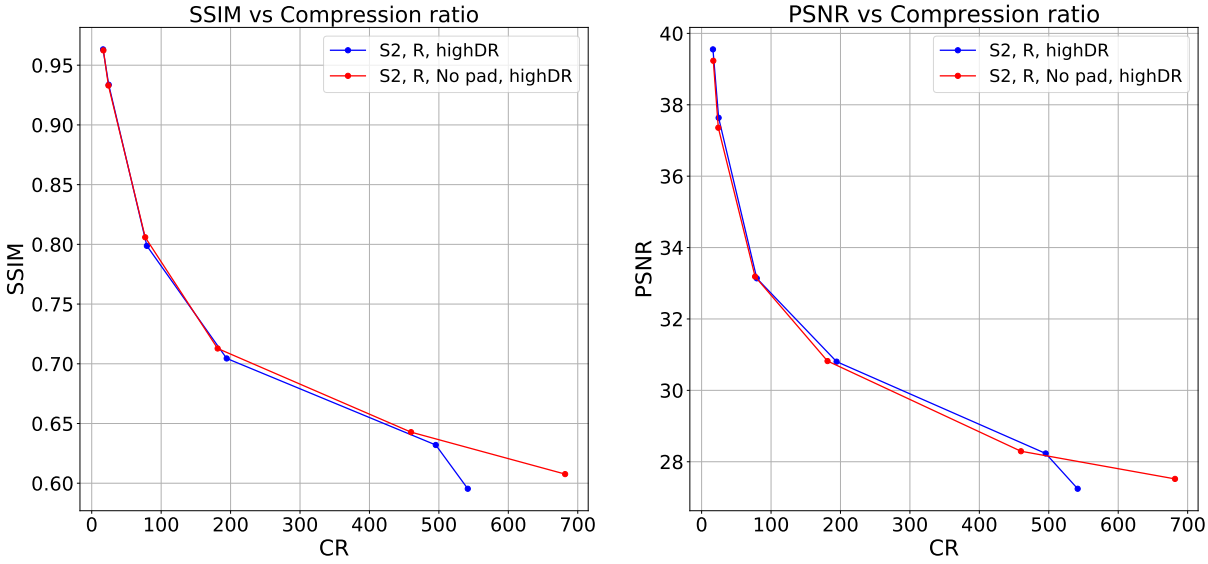

Figure 4.6 Comparative of the SSIM and PSNR metrics between the two possible padding setups

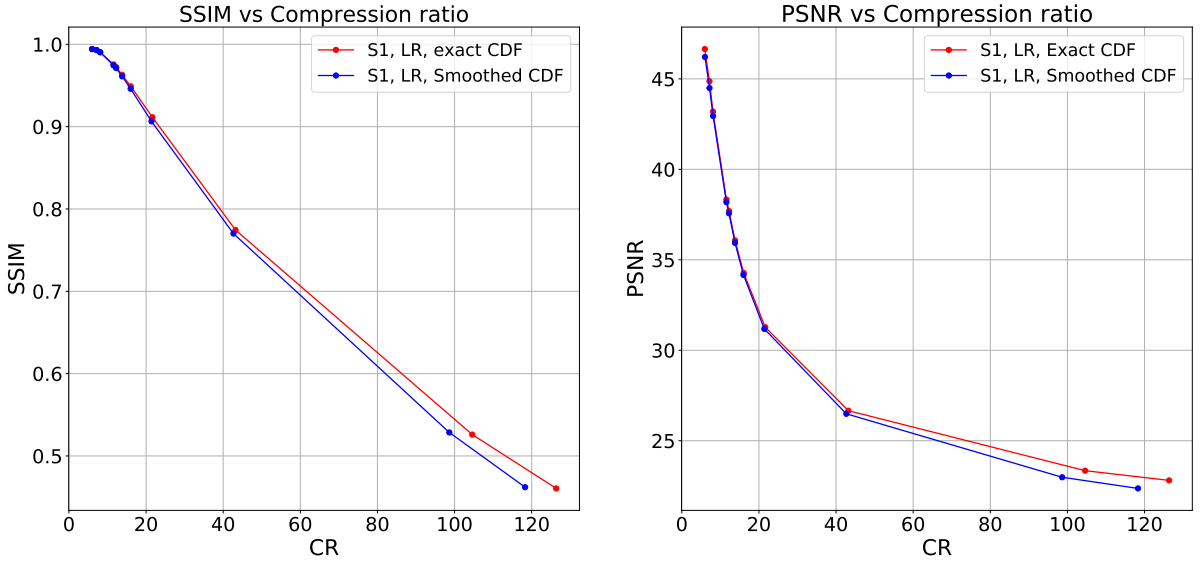

Figure 4.7 Comparative between the usage of a smoothed CDF and an exact CDF to perform the entropy lossless compression on the quantized values

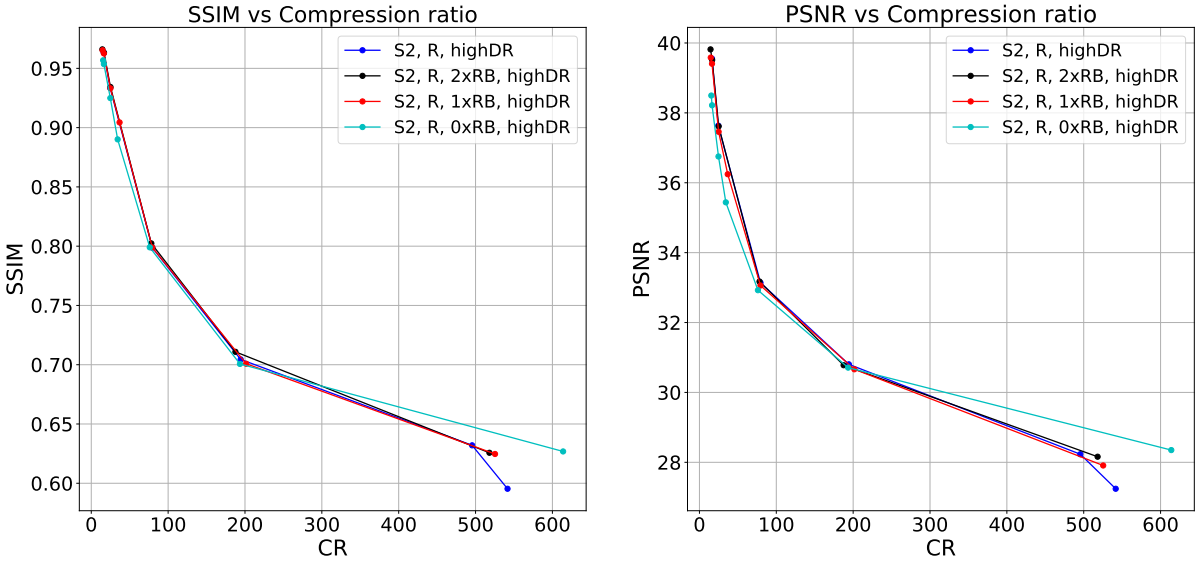

Figure 4.8 Comparison between setting 0 to 3 residual blocks on the network

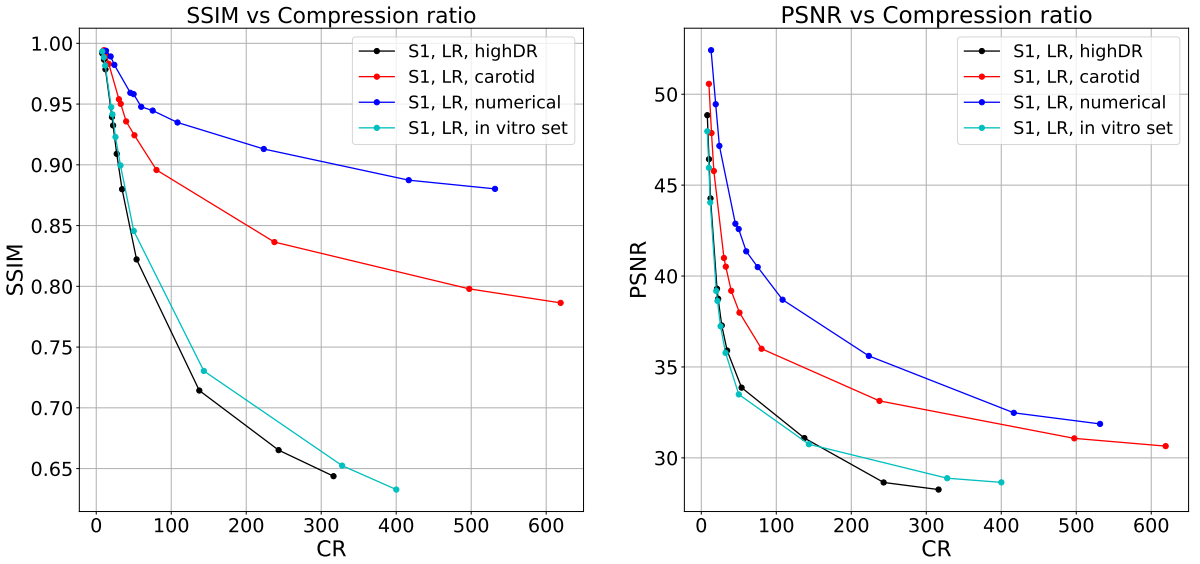

Figure 4.9 Comparison between the carotid, *in vitro*, and numerical test datasets

4.4.1 US image assessment

After processing the US raw data, the RF and B-mode images are obtained. The B-mode images will have a maximum value of 0 dB, and are clipped at -60 dB before plotting and obtaining their metrics. The SSIM, PSNR and compression ratio values are averaged from each set of images. The ordinates axis represents the depth in the media, while the abscissa axis represents the width. As observed (specially in the zoomed images), the areas where the intensity is lower, the distortion is also larger. Meanwhile, the areas with larger intensity show less-modified patters, where in some cases the distortion/pattern-modifications are imperceptible.

In Figure 4.21, the metrics of the US raw data are compared against the metrics of the US RF images. Due to the delay and sum processing, the incoherent noise is summed and therefore it is expected to

have slightly better metrics in the RF images than in the US raw data.

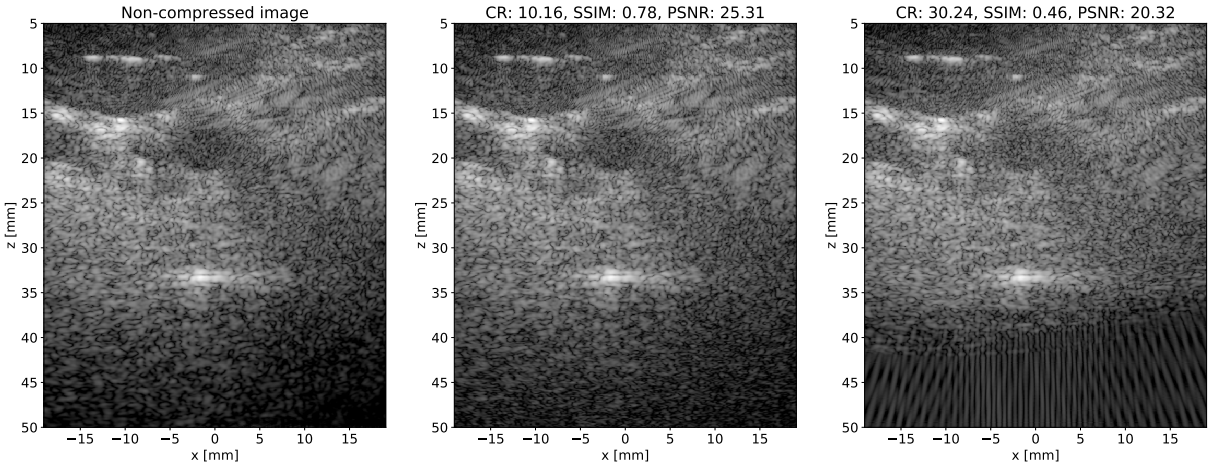

Figure 4.10 Carotid B-mode image 1

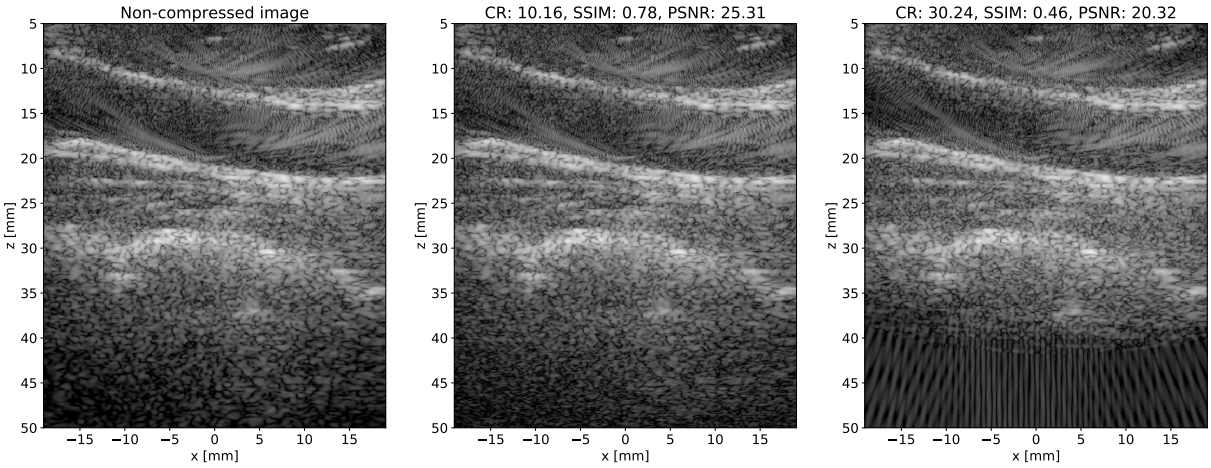

Figure 4.11 Carotid B-mode image 2

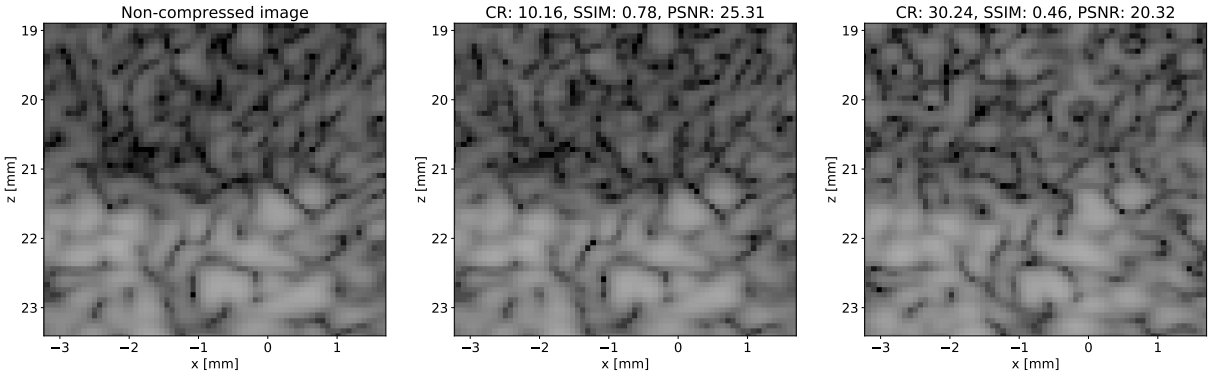

Figure 4.12 Zoom in carotid B-mode image 1

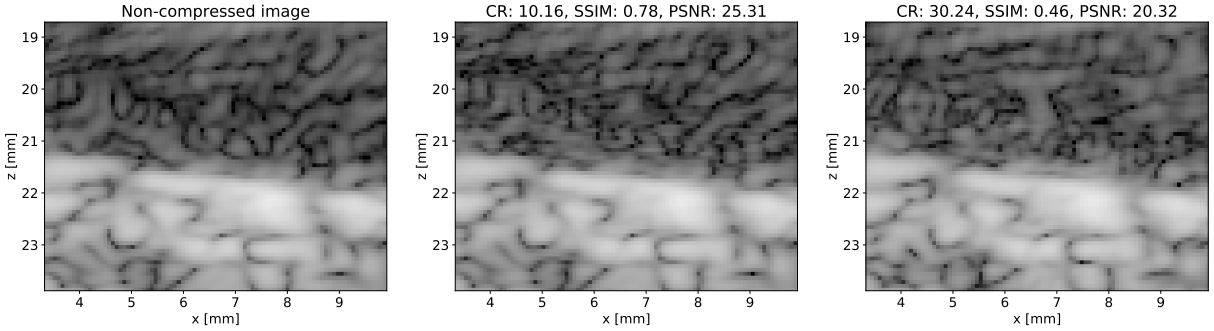

Figure 4.13 Zoom in carotid B-mode image 2

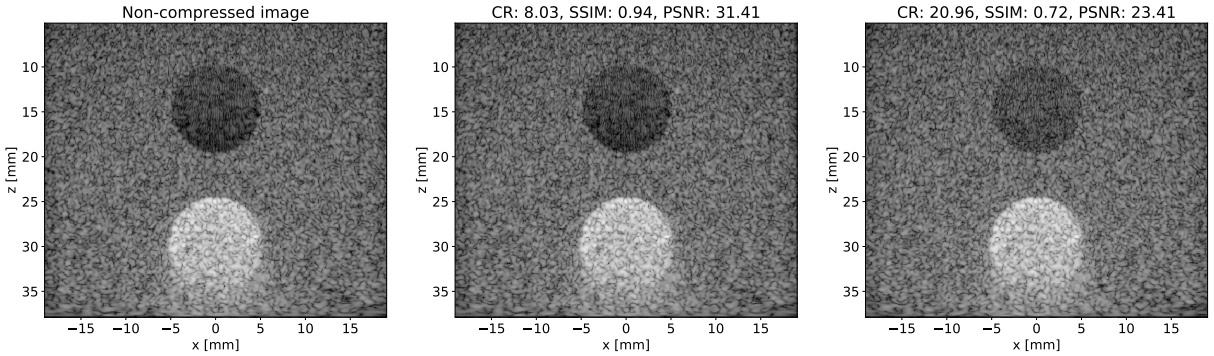

Figure 4.14 High dynamic range B-mode image

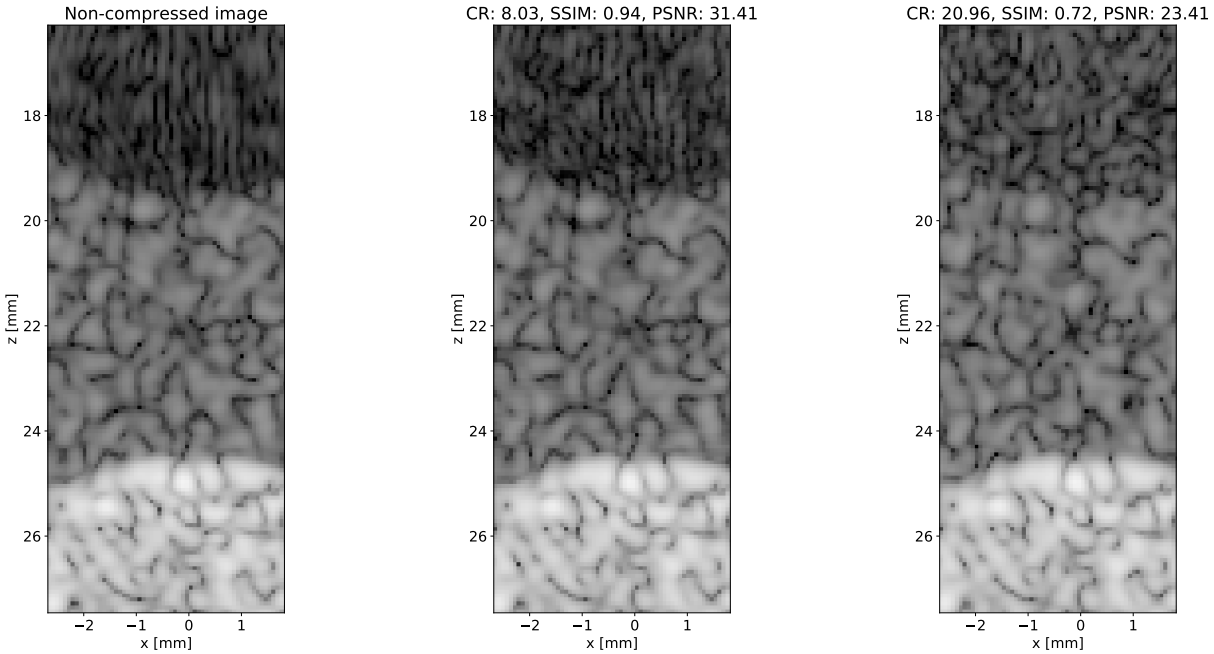

Figure 4.15 Zoom in high dynamic range B-mode image

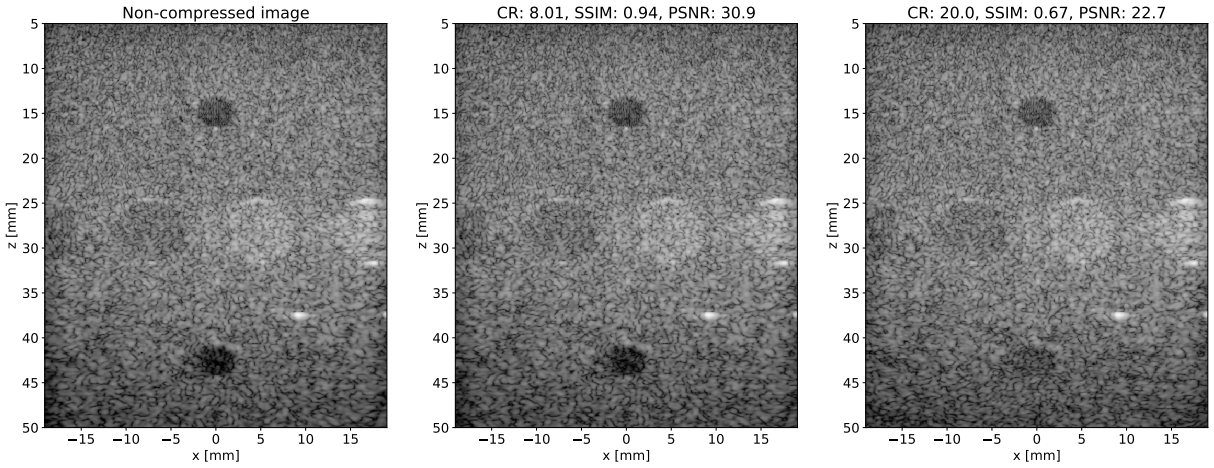

Figure 4.16 *in vitro* B-mode image 1

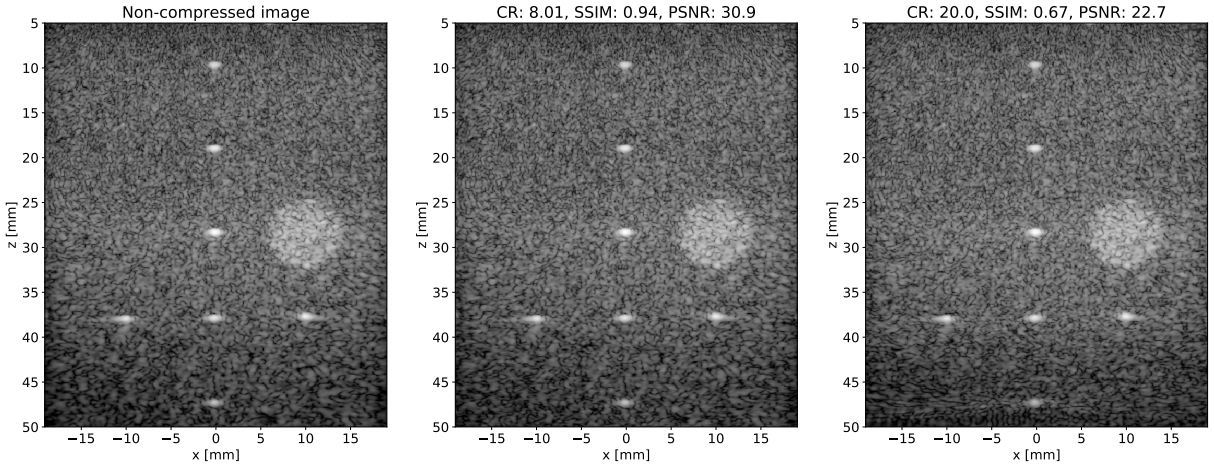

Figure 4.17 *in vitro* B-mode image 2

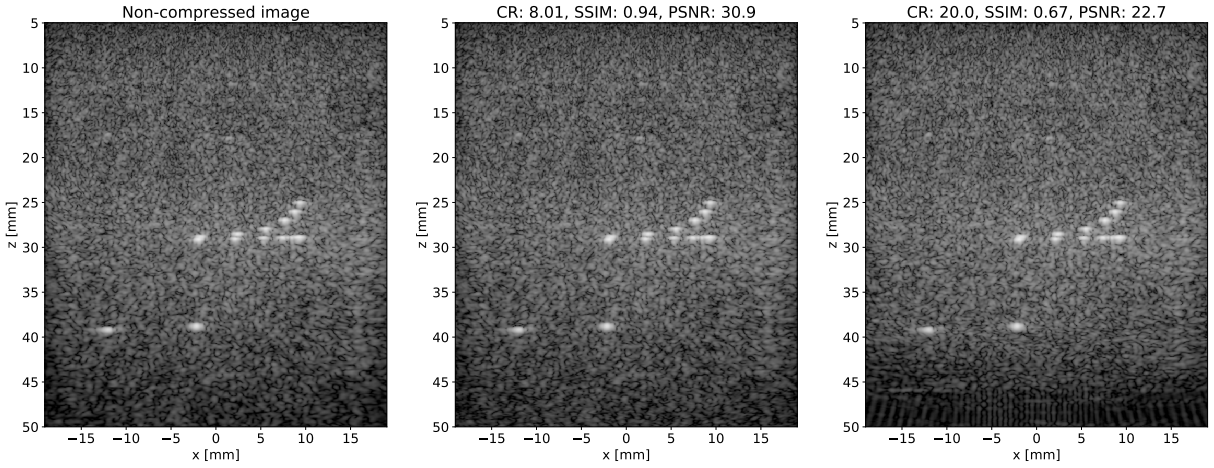

Figure 4.18 *in vitro* B-mode image 3

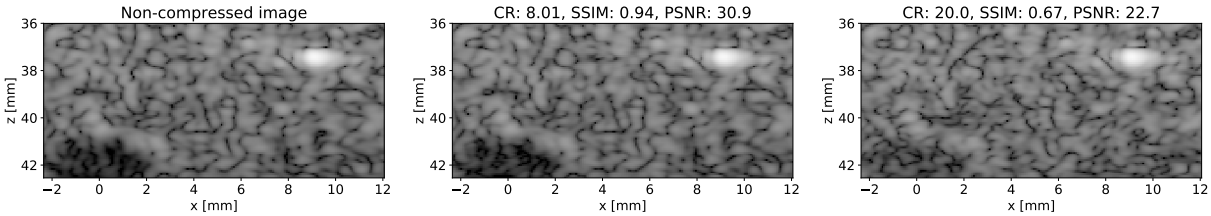

Figure 4.19 Zoom in *in vitro* B-mode image 1

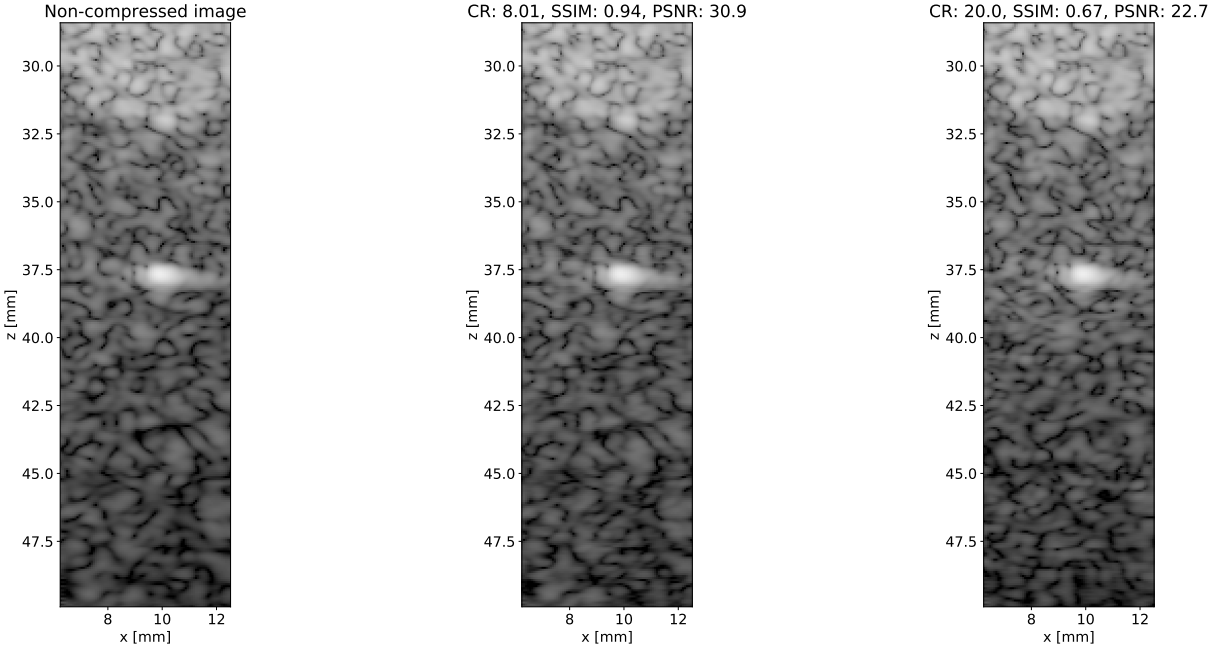

Figure 4.20 Zoom in *in vitro* B-mode image 2

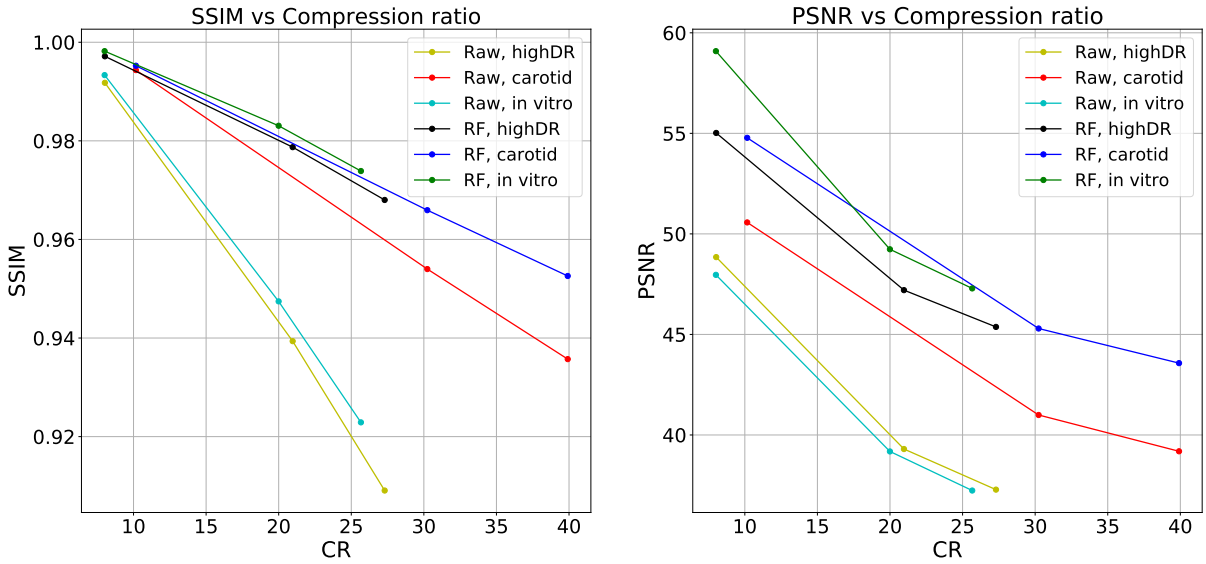

Figure 4.21 Comparison of SSIM and PSNR metrics between the Raw US data and the RF US images

4.4.2 Data-rate reduction

In Table 4.2 a set of expected data-rates are shown for different frequencies. The values are computed taking into account the 128 sensors, 1024 samples of US data per image, and 16 bits per values. Therefore, the data rate is computed as in Equation (4.3).

$$Rate[bit/s] = 128[sensor] \times 1024[sample/sensor] \times 16[bit/sample] \times frequency[s^{-1}] \tag{4.3}$$

The compression ratios that provide SSIM values on the high-dynamic-range US data over 0.9 are approximately between 8 and 30, which are the approximate compression ratios for the high-dynamic-range test dataset (considered the worst case). Taking into account the previous compression ratios, the resulting data rates for compression ratios of 8 and 30 are shown in the low compression data-rate and high compression data-rate respectively.

Table 4.2 Data-rates computed for different frequencies and compression ratios

Frequency[kHz]	Rate[Gb/s]	Low compression Rate[Gb/s]	High compression Rate[Gb/s]
5	10.5	1.3	0.35
7.5	15.7	1.9	0.52
10	21	2.6	0.7

4.4.3 Data transmission over different communication channels

In this subsection the data rates computed in Section 4.4.2 are compared with the data rates that USB and wireless interfaces can provide.

USB 3.0 interface

In the USB 3.0 specification [20], a rate of 400 MBps (3.2 Gbps) is considered a realistic raw data throughput. Taking into account that 3.2 Gbps is a realistic data-rate value, all the data-rates obtained for different frequencies and compression ratios in Table 4.2 may be transmitted through a USB 3.0 interface without trouble. In fact, a compression ratio of approximately 6.6 would be enough to be below the USB 3.0 capacity for the 10 kHz frequency.

Wireless interface

Regarding the usage of a wireless interface; considering the different versions of the WiFi protocol (802.11), the most suitable one may be the 802.11ad. This version of the protocol targets transfer rates of several Gbps in short distances, within the 60 GHz band. High data rates can be achieved due to the higher frequency band, as well as to the lower congestion compared to the 2.4 GHz and 5 GHz bands commonly used in other versions of the protocol. Since the data transfer of a wireless US device may be performed in general, in short distances, this version of the protocol may be suitable. As commented in [21], 802.11ad supports data-rates of 4.7 Gbps within a walled-in area or open space. Therefore, it may also be a suitable protocol to transfer the compressed US data for all the data rates computed in Table 4.2.

5 Conclusions and perspectives

The results obtained from the tests performed show that the current model can be used to perform lossy compression on US data. The compression rates obtained for SSIM values on the raw US data of 0.9 is approximately 30, while for SSIM values on the raw US data of 0.99 the compression rate obtained is around 8. From those values, a range of data-rates has been computed for different frequencies and compression ratios. The resulting data-rates after compressing the US data are clearly below the maximum data rates that current USB and wireless protocols can support. Therefore, in terms of data compression and data-rate reduction metrics the model implemented may be suitable. The compression ratios are much larger than the ones obtained by using a lossless compression method such as the ones obtained in [22], where the compression ratios obtained are below 1.5 (but lossless and real-time). The results on the images obtained from the US decompressed data show that the SSIM is reduced in comparison to the SSIM in the raw US data and the RF images, specially for high compression ratios. Furthermore, the effects of the distortion mainly appear in the areas of the images where the intensity is lower.

Regarding the model; it has been found that the effect of the residual blocks of the original network is imperceptible on the decompressed data' s quality. As well as that, the number of channels (depth) along the network has also been reduced and the results obtained are very similar(between S1 and S2 filter setups), while reducing the inference time. The main difference that can be appreciated when reducing the number of channels, is the maximum reachable quality of the model. For qualities lower than the maximum, the qualities are similar between a given model, and another model with more filters.

Regarding future perspectives; further modifications could be tested in the current model in order to improve the results in terms of rate-distortion trade-offs; that is, reaching better qualities for the current data rates. In addition, the current software implementation may be adapted in order to target real-time applications.

Bibliography

[1] L. Theis, W. Shi, A. Cunningham, and F. Huszár, *Lossy image compression with compressive autoencoders*, 2017. eprint: arXiv:1703.00395.

[2] J. Ballé, V. Laparra, and E. P. Simoncelli, *End-to-end optimized image compression*, 2016. eprint: arXiv:1611.01704.

[3] J. Ballé, V. Laparra, and E. P. Simoncelli, "End-to-end optimization of nonlinear transform codes for perceptual quality", in *2016 Picture Coding Symposium (PCS)*, 2016, pp. 1–5.

[4] O. Rippel and L. Bourdev, *Real-time adaptive image compression*, 2017. eprint: arXiv:1705.05823.

[5] M. Li, W. Zuo, S. Gu, D. Zhao, and D. Zhang, *Learning convolutional networks for content-weighted image compression*, 2017. eprint: arXiv:1703.10553.

[6] V. Dumoulin and F. Visin, *A guide to convolution arithmetic for deep learning*, 2016. eprint: arXiv:1603.07285.

[7] J. Ballé, V. Laparra, and E. P. Simoncelli, *End-to-end optimization of nonlinear transform codes for perceptual quality*, 2016. eprint: arXiv:1607.05006.

[8] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back propagating errors", *Nature*, vol. 323, pp. 533–536, 1986.

[9] C. E. Shannon, "A mathematical theory of communication", *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 5, pp. 3–55, 2001.

[10] G. N. N. Martin, "Range encoding: An algorithm for removing redundancy from a digitised message", *Video Data Recoding Conference, Southampton*, vol. 1979, 1979.

[11] Wikipedia contributors, *Range encoding — Wikipedia, the free encyclopedia*, [Online; accessed 31-December-2018], 2018.

[12] *MATLAB image pad documentation*, https://ch.mathworks.com/help/vision/ref/imagepad.html, Accessed: 2019-01-03.

[13] K. Oku, *Range coder implementation*, https://github.com/kazuho/rangecoder, Accessed: 2019-01-04.

[14] L. Theis, *Range coder python interface implementation*, https://pypi.org/project/range-coder/, Accessed: 2019-01-04.

[15] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. eprint: arXiv:1412.6980.

[16] *Tensorflow guide*, https://www.tensorflow.org/guide/, Accessed: 2019-01-06.

[17] *Kodak image dataset*, http://r0k.us/graphics/kodak/, Accessed: 2019-01-07.

[18] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity", *IEEE Transactions on Image Processing*, vol. 13, pp. 600–612, 2004.

[19] *PICMUS evaluation framework*, `https://www.creatis.insa-lyon.fr/EvaluationPlatform/picmus/index.html`, Accessed: 2019-01-16.

[20] *Universal serial bus 3.0 specification*, `https://www.usb3.com/whitepapers/USB%203%200%20(11132008)-final.pdf`, Accessed: 2019-01-16.

[21] *Wi-fi industry adopts 802.11ad for high performance*, `https://www.qualcomm.com/media/documents/files/strategy-analytics-report-wi-fi-industry-adopts-802-11ad-for-high-performance-.pdf`, Accessed: 2019-01-17.

[22] J. H. Kim, S. Yeo, J. W. Kim, K. Kim, T.-K. Song, Ch. Yoon, *et al.*, "Real-time lossless compression algorithm for ultrasound data using BL universal code", *Sensors*, vol. 18, 2018.